Wasm-mutate: Fuzzing WebAssembly Compilers with E-Graphs

Talk proposal for EGRAPHS 2022.

Javier Cabrera Arteaga
KTH Royal Institute of Technology
javierca@kth.se

Nicholas Fitzgerald
Fastly Inc.
nfitzgerald@fastly.com

Martin Monperrus
KTH Royal Institute of Technology
monperrus@kth.se

Benoit Baudry
KTH Royal Institute of Technology
baudry@kth.se

1 Introduction

Fuzzing is a software testing technique used to find security, stability, and correctness issues by feeding pseudo-random data as input to a program. [2]. A program's input domain is typically too large to enumerate exhaustively. Therefore, instead of exhaustive enumeration, we use fuzzing to test a subset of the input space. For example, the input space of compilers is all possible programs that can be written in the compiler's source language. That is an infinite value space, which needs to be sampled when searching for defects in a compiler. In this work, we use e-graphs to improve input generation for fuzzing compilers. We implement this technique for fuzzing Wasm compilers, but the approach should also generalize to other source languages.

We present, wasm-mutate [1] , a tool for fuzzing Wasm compilers and other Wasm related tools such as validators or interpreters. Given a valid Wasm input program, it produces a sequence of Wasm programs that are semantically equivalent to its original input. Wasm-mutate follows the notion of program equivalence modulo input [1], i.e., for all possible program inputs the variants should provide the same output. It represents the search space for new variants as an e-graph, and it exploits the property that any traversal through the e-graph represents a semantically equivalent variant of the input program.

This talk will focus on the proposed algorithm to traverse the e-graph in order to provide semantically equivalent code variants, as well as the ways we leverage wasm-mutate while fuzzing. In the following sections, we discuss the design behind wasm-mutate and its e-graph traversal algorithm.

2 The Peephole Mutator's E-Graph

At a high level, wasm-mutate is a framework for defining mutators. It takes a Wasm module as input and returns a lazy sequence of mutated variants of that input Wasm. The primary mutator is the peephole mutator, which is responsible for rewriting instruction sequences in the input Wasm's function bodies. The peephole mutator 1.) inspects the Wasm's code section, 2.) selects a random function within it, 3.) selects a random instruction within the function, 4.) constructs a data-flow graph rooted at the selected instruction, 5.) applies one or more random rewrite rules, and 6.) re-encodes the Wasm module with the new, rewritten expression to produce a mutated variant. The peephole mutator defines a number of rewrite rules. A rewrite rule can be seen as a pair, `(LHS, RHS)`, in which the `LHS` part is the code to replace and `RHS` is the semantically equivalent replacement. For example, $(x,\ x\ or\ x)$, in which the x `LHS` is replaced by an idempotent bit-wise *or* operation with itself.

wasm-mutate's performance is essential, as fuzzing campaigns often have a time budget and the faster we can generate test inputs, the more we can exercise our compiler within the time budget. At the same time, the quality of the generated test inputs cannot be compromised: quickly testing the same low-quality input over and over is inefficient and unlikely to uncover bugs.

E-graphs, and the egg [4] implementation, are useful for balancing performance and quality of generated test inputs. Although e-graphs are often paired with a cost function to extract optimal expressions for program optimizers, any path traversal through the e-graph produces a semantically equivalent expression. Wasm-mutate leverages this property to generate mutated variants. We traverse the e-graph, randomly selecting an e-node within each e-class that we visit, producing a random-but-equivalent expression. Because the e-graph typically represents an infinite number of expressions, we can extract an infinite number of them that are equivalent to the input expression from the e-graph. Although the construction of the e-graph itself can be expensive, we can reuse the e-graph many times after construction, amortizing the construction costs. We can also speed up construction by limiting the number of rewrites applied to the e-graph. Extracting a random expression from the e-graph is described by algorithm 1.

The e-graph traversal is summarized in Algorithm 1. It receives an e-graph, an e-class (initially the root's e-class), and the maximum depth of expression to extract. The depth

[1] https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate

PL'18, January 01–03, 2018, New York, NY, USA
2018.

Algorithm 1 e-graph traversal algorithm.

```
1:  procedure TRAVERSE(egraph, eclass, depth)
2:      if depth = 0 then
3:          return smallest_tree_from(egraph, eclass)
4:      else
5:          nodes ← egraph[eclass]
6:          node ← random_choice(nodes)
7:          expr ← (node, operands = [])
8:          for each child ∈ node.children do
9:              subexpr ← TRAVERSE(egraph, child, depth − 1)
10:             expr.operands ← expr.operands ∪ {subexpr}
11:         return expr
```

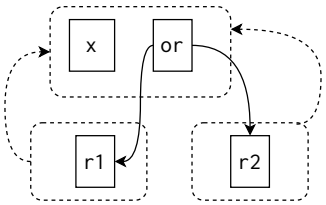

Figure 1. e-graph for idempotent bitwise-or rewriting rule. Solid lines represent operand-operator relations and dashed lines represent equivalent class inclusion.

parameter ensures that we don't infinitely recurse, and as soon as it becomes zero, the algorithm returns the smallest expression out of the current e-class (line 3). Otherwise, we select a random e-node from the e-class (lines 5 and 6), and the process recursively continues with the children of the selected e-node (line 8) with a decreasing depth. The subexpressions are composed together (line 10) for each child, and then the entire expression is returned (line 11). The contribution of wasm-mutate is what is defined between lines 5 and 11.

We illustrate the algorithm with an example using the same rewriting rule previously mentioned. The peephole mutator randomly selects an instruction. An e-graph is constructed for the instruction. In this example, the peephole mutator has only one rewriting rule:

$$x \rightarrow (x\ or\ \ x)$$

Notice that this rewrite rule is not optimal: it increases the Wasm code size, rather than decreasing it, and introduces an additional operation to be evaluated. However, in the context of fuzzing, it is valuable. This rewrite rule can stress a compiler's constant folding optimization passes.

The e-graph for our single rewrite rule is shown in Figure 1. Applying the procedure of the algorithm 1 will provide, in theory, an infinite number of equivalent expressions:

$$depth = 0 \rightarrow x$$
$$depth = 1 \rightarrow x\ or\ x$$
$$depth = 2 \rightarrow x\ or\ (x\ or\ x)$$
$$depth = 3 \rightarrow (x\ or\ x)\ or\ (x\ or\ (x\ or\ x))$$

3 Fuzzing with wasm-mutate

We have used wasm-mutate in two different ways while fuzzing the Wasmtime[2] WebAssembly runtime :

3.1 Differential Fuzzing

In this mode of fuzzing [3], we run an initial Wasm program with some arbitrary inputs, then generate a series of mutated variants with wasm-mutate and run each variant with the same inputs. Because wasm-mutate produces variants that are equivalent modulo input, if the results of the variants are not identical to the results of the initial Wasm program, that is indicative of a bug in Wasmtime or its compiler.

3.2 Structure-Aware Fuzzing with a Custom libFuzzer Mutator

In this mode of fuzzing, we relax wasm-mutate's equivalent modulo input constraint and allow it to generate variants that will not produce the same results that the original Wasm program would. We hook it up to libFuzzer's custom mutation hook[3] and use it as a strategy to generate new, independent Wasm test cases. libFuzzer's built in mutators will do things like add, remove, or swap bytes from a file in the fuzz corpus. These canned mutations will most often produce an invalid Wasm file. On the other hand, wasm-mutate will always produce valid Wasm files, avoiding us wasting fuzzing cycles on uninteresting inputs that will bounce off our Wasm compiler's parser, or which might parse but will contain type errors and fail to validate.

4 Conclusion

Using e-graphs to generate semantically equivalent programs and using those programs for differential fuzzing is a promising technique. While we have used it to exercise a WebAssembly compiler and runtime, the approach should generalize to other source languages.

References

[1] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. 216–226.

[2] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (dec 2018), 1–13. https://doi.org/10.1186/S42400-018-0002-Y/TABLES/5

[3] William M. McKeeman. 1998. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL* 10, 1 (1998), 100–107.

[4] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2020. egg: Fast and Extensible Equality Saturation. *arXiv e-prints*, Article arXiv:2004.03082 (April 2020), arXiv:2004.03082 pages. arXiv:2004.03082 [cs.PL]

[2] https://github.com/bytecodealliance/wasmtime

[3] https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md